Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases

Esther Pacitti, Pascale Minet, Eric Simon

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases

Esther Pacitti, Pascale Minet and Eric Simon

No 3654

Avril 1999

THÈMES 3 et 1

ISSN 0249-6399

Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases

Esther Pacitti, Pascale Minet and Eric Simon *

Thèmes 3 et 1 — Interaction homme-machine,
images, données, connaissances — Réseaux et systèmes
Projets RODIN et REFLECS

Rapport de recherche n ° 3654 — Avril 1999 — 35 pages

Abstract: In a lazy master replicated database, a transaction can commit after updating one replica copy (primary copy) at some master node. After the transaction commits, the updates are propagated towards the other replicas (secondary copies), which are updated in separate refresh transactions. A central problem is the design of algorithms that maintain replica's consistency while at the same time minimizing the performance degradation due to the synchronization of refresh transactions. In this paper, we propose a simple and general refreshment algorithm that solves this problem and we prove its correctness. The principle of the algorithm is to let refresh transactions wait for a certain "deliver time" before being executed at a node having secondary copies. We then present two main optimizations to this algorithm. One is based on specific properties of the topology of replica distribution across nodes. In particular, we characterize the nodes for which the deliver time can be null. The other improves the refreshment algorithm by using an immediate update propagation strategy. Our performance evaluation demonstrate the effectiveness of this optimization.

Key-words: replicated databases, lazy master, replica consistency, refreshment algorithm, data mart, data warehouse.

(Résumé : tsvp)

* {Esther.Pacitti}{Pascale.Minet}{Eric.Simon}@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : 01 39 63 55 11 - International : +33 1 39 63 55 11
Télécopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

Algorithmes de Maintien de la Cohérence dans des Bases de Données Répliquées Gérées selon un Modèle Asymétrique

Résumé : Dans une base de données où les données répliquées sont gérées selon le modèle asymétrique, une transaction peut valider après avoir mis à jour une seule réplique (la copie primaire) sur le noeud maître. Les mises à jour sont ensuite propagées aux autres répliques (les copies secondaires), qui sont mises à jour dans des transactions de rafraîchissement séparées. La conception d'algorithmes chargés de maintenir la cohérence des répliques tout en minimisant la dégradation des performances due à la synchronisation des transactions de rafraîchissement, est un problème crucial. Dans ce rapport, nous proposons un algorithme de rafraîchissement simple et général qui résout ce problème, et nous prouvons sa correction. Le principe de l'algorithme est de faire attendre, les transactions de rafraîchissement, avant de les exécuter sur un noeud ayant des copies secondaires. Nous présentons ensuite deux optimisations majeures de cet algorithme. La première optimisation est basée sur des propriétés topologiques de la configuration des répliques. En particulier, nous caractérisons les noeuds pour lesquels, il est inutile d'attendre. La deuxième optimisation améliore la fraîcheur des données en utilisant une stratégie de propagation immédiate des mises à jour. Notre évaluation de performances montre l'efficacité de cette optimisation.

Mots-clé : bases de données répliquées, mode asymétrique, cohérence de données répliquées, algorithme de rafraîchissement, marché de données, entrepôt de données.

1 Introduction

Lazy replication (also called asynchronous replication) is a widespread form of data replication in (relational) distributed database systems [23]. With lazy replication, a transaction can commit after updating one replica copy at some node[1]. After the transaction commits, the updates are propagated towards the other replicas, and these replicas are updated in separate refresh transactions. In this paper, we focus on a specific lazy replication scheme, called *lazy master* replication [15] (also called Single-Master-Primary-Copy replication in [4]). There, one replica copy is designated as the *primary copy*, stored at a *master* node, and update transactions are only allowed on that replica. Updates on a primary copy are distributed to the other replicas, called *secondary copies*. A major virtue of lazy master replication is its ease of deployment [4, 15]. In addition, lazy master replication has gained considerable pragmatic interest because it is the most widely used mechanism to refresh data warehouses and data marts in data warehousing architectures [6, 23].

However, lazy master replication may raise a consistency problem between replicas. Indeed, an observer of a set of replica copies at some node at time t may see a state I of these copies that can never be seen at any time, before or after t, by another observer of the same copies at some other node. We shall say that I is an *inconsistent* state. As a first example, suppose that two data marts S_1 and S_2 both have secondary copies of two primary copies stored at two different data source nodes[2]. If the propagation of updates coming from different transactions at the master nodes is not properly controlled, then refresh transactions can be performed in a different order at S_1 and S_2, thereby introducing some inconsistencies between replicas. These inconsistencies in turn can lead to inconsistent views that are later almost impossible to reconciliate [17].

Let us expand the previous example into a second example. Suppose that a materialized view V of S_1, considered as a primary copy, is replicated in data mart S_2. Now, additional synchronization is needed so that the updates issued by the two data source nodes *and* the updates of V issued by S_1 execute in the same order for all replicas in S_1 and S_2.

Thus, a central problem is the design of algorithms that maintain replica's consistency in lazy master replicated databases, while at the same time minimizing the performance degradation due to the synchronization of refresh transactions. Considerable attention has been given to the maintenance of replicas' consistency in a lazy replicated system. First, many papers addressed this problem in the context of lazy group replicated systems, which require the reconciliation of updates coming from multiple primary copies [25, 13, 15, 1, 28]. Some papers have proposed to use weaker consistency criterias that depend on the application semantics. For instance, in the OSCAR system [8], each node processes the updates received from master nodes according to a specific weak-consistency method that

[1] From now on, we suppose that replicas are relations.

[2] This frequent situation typically arises when no corporate data warehouse has been set up between data sources and data marts. Quite often, each data mart, no matter how focused, ends up with views of the business that overlap and conflict with views held by other data marts (e.g., sales and inventory data marts). Hence, the same relations can be replicated in both data marts [17].

is associated with each secondary copy. However, their proposition does not yield the same notion of consistency as ours. In [2, 26, 3], authors propose some weak consistency criterias based on time and space, e.g., a replica should be refreshed after a time interval or after 10 updates on a primary copy. There, the concern is not anymore on fast refreshment and hence these solutions are not adequate to our problem. In [7], the authors give conditions over the placement of secondary and primary copies into sites under which a lazy master replicated database can be guaranteed to be globally serializable (which corresponds to our notion of consistency). However, they do not propose any refreshment algorithm for the cases that do not match their conditions, such as our two previous examples. Finally, some synchronization algorithms have been proposed and implemented in commercial systems, such as Digital's Reliable Transaction Router [4], where the refreshment of all secondary copies of a primary copy is done in a distributed transaction. However, to the best of our knowledge, these algorithms do not assure replica consistency in cases like our second above example.

This paper makes three important contributions with respect to the central problem mentionned before. First, we analyze different types of configurations of a lazy master replicated system. A configuration represents the topology of distribution of primary and secondary copies accross the system nodes. It is a directed graph where a directed arc connects a node N to a node N' if N holds a primary copy of some secondary copy in N'. We formally define the notion of correct refreshment algorithm that assures database consistency. Then, for each type of configuration, we define sufficient conditions that must be satisfied by a refreshment algorithm in order to be correct. Our results considerably generalize already published results such as [7].

As a second contribution, we propose a simple and general refreshment algorithm, which is proved to be correct for a large class of acyclic configurations (including for instance, the two previous examples). We show how to implement this algorithm using system components that can be added to a regular database system. Our algorithm makes use of a reliable multicast with a known upper bound, that preserves a global FIFO order. Our algorithm also uses a deferred update propagation strategy, as offered by all commercial replicated database systems. The general principle of the algorithm is to make every refresh transaction wait a certain "deliver time" before being executed.

As a third contribution, we propose two main optimizations to this algorithm. First, using our correctness results on types of configurations, we provide a static characterization of the nodes for which the deliver time can be null. Second, we give an optimized version of the algorithm that uses an immediate update propagation strategy, as defined in [24]. We give a performance evaluation based on simulation that demonstrates the value of this optimization by showing that it significantly improves the freshness of secondary copies.

The rest of this paper is structured as follows. Section 2 introduces our lazy master replication framework, and the typology of configurations. Section 3 defines the correctness criteria for each type of configuration. Section 4 describes our refreshment algorithm, how to incorporate it in the system architecture of nodes, and proves its correctness. Section

5 presents our two main optimizations. Section 6 introduces our simulation environment and presents our performance evaluation, which shows the effectiveness of our optimization. Section 7 discusses some related work. Finally, Section 8 concludes. Section 9 is the appendix containing the proofs of propositions given in Section 3.

2 Lazy Master Replicated Databases

We define a (relational) lazy replicated database system as a set of n interconnected database systems, henceforth called *nodes*. Each node N_i hosts a relational database whose schema consists of a set of pairwise distinct relational schemas, whose instances are called relations. A replication scheme defines a partitioning of all relations of all nodes into partitions, called *replication sets*. A replication set is a set of relations having the same schema, henceforth called *replica copies*[3]. In this section, we define a special class of replicated systems, called *lazy master*, which constitutes our framework.

2.1 Ownership

Following [15], the *ownership* defines the node capabilities for updating replica copies. In a replication set, there is a single updatable replica copy, called *primary* copy (denoted by a capital letter), and all the other relations are called *secondary* copies (denoted by lower-case letters). We assume that a node never holds the primary copy and a secondary copy of the same replication set. We distinguish between three kinds of nodes in a lazy master replicated system.

Definition 2.1 *(Types of nodes).*

1. *A node M is said to be a* master *node iff : $\forall m \in M$ m is a primary copy.*
2. *A node S is said to be a* slave *node iff : $\forall s \in S$ s is a secondary copy of a primary copy of some master node.*
3. *A node MS is said to be a* master/slave *node iff : $\forall ms \in MS$, either ms is a secondary copy, or ms is a primary copy.*

Finally, we define the following slave and master dependencies between nodes. A master or master/slave node M is said to be a *master node of* a slave or master/slave node S iff there exists a secondary copy r in S of a primary copy R in M. We also say that S is a *slave node of* M.

[3] A replication set can be reduced to a singleton if there exists a single copy of a relation in the replicated system.

2.2 Configurations

Slave dependencies define a DAG, called configuration.

Definition 2.2 *(Configuration). A* configuration *of a replicated system is defined by a directed graph, whose nodes are the nodes of the replicated system, and there is a directed arc from a node N to a node N' iff N' is a slave node of N. Node N is said to be a predecessor of node N'.*

In the following, we distinguish different types of configurations. The rationale for these configurations will become clear later. Intuitively, to each configuration will correspond a correctness criteria to guarantee database consistency. In the figures illustrating the configurations, we use integers to represent nodes in order to avoid confusion with the names of the relations that are displayed as annotation of nodes.

Definition 2.3 *(1master-per-slave configuration). An acyclic configuration in which each node has at most one predecessor is said to be a* 1master-per-slave *configuration.*

This configuration, illustrated in Figure 1(a), corresponds to a "data dissemination" scheme whereby a set of primary copies of a master or master/slave node is disseminated towards a set of nodes. This configuration characterizes for instance the case of several data marts built over a centralized corporate data warehouse.

Definition 2.4 *(1slave-per-master configuration). An acyclic configuration in which each node has at most one successor is said to be a* 1slave-per-master *configuration.*

This configuration, illustrated in Figure 1(b), corresponds to what is often called a "data consolidation" scheme, whereby primary copies coming from different nodes are replicated into a single node. Such a configuration characterizes for instance a configuration wherein a data warehouse node (or even, an operational data store node) holds a set of materialized views defined over a set of relations stored by source nodes. In this context, replicating the source relations in the data warehouse node has two main benefits. First, one can take advantage of the replication mechanism to propagate changes from the source towards the data warehouse. Second, it assures the self-maintainability of all materialized views in the data warehouse, thereby avoiding the problems mentioned in [30].

Definition 2.5 *(bowtie configuration). An acyclic configuration in which there exist two distinct replicas X_1 and X_2 and four distinct nodes M_1, M_2, S_1 and S_2 such that (i) M_1 holds the primary copy of X_1 and M_2 the primary copy of X_2, and (ii) both S_1 and S_2 hold secondary copies of both X_1 and X_2.*

Such configuration, illustrated in Figure 1(c), generalizes the two previous configurations by enabling arbitrary slave dependencies between nodes. This configuration characterizes, for instance, the case of several data marts built over several data sources. The benefits of a replication mechanism are the same as for a data consolidation configuration.

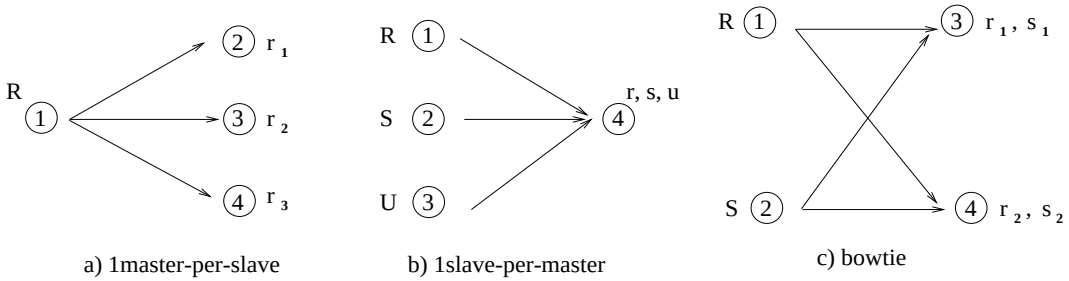

Figure 1: Examples of Configurations

Definition 2.6 *(triangular configuration). An acyclic configuration in which there exist three distinct nodes M, MS and S such that (i) MS is a successor of M, and (ii) S is a successor of both M and MS, is said to be a* triangular *configuration. Nodes M, MS and S are said to form a* triangle.

This configuration, illustrated in Figure 2(a), slightly generalizes the two first configurations by enabling a master/slave node to play an added intermediate role between a master node and a slave node.

Definition 2.7 *(materialized view). A primary copy of a master/slave node MS which is defined as the result of the query over a set of secondary copies of MS is called a* materialized view.

Definition 2.8 *(view triangular configuration). A derived configuration in which all the primary copies hold by any node MS of any triangle are materialized views of local secondary copies, is said to be a* view triangular *configuration.*

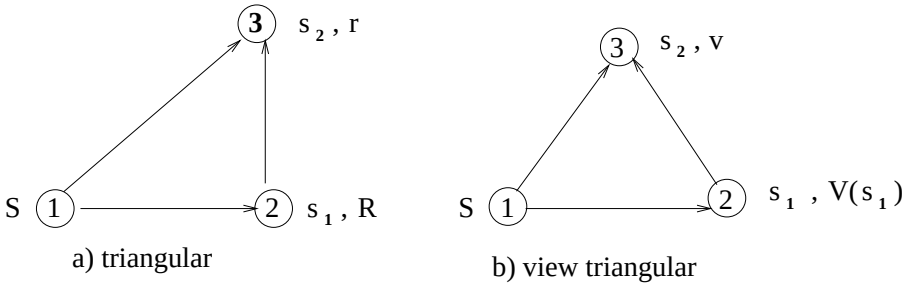

Figure 2: Examples of Configurations

This configuration, illustrated in Figure 2(b), characterizes, for instance, the case of two independent data marts defined over the same data warehouse in which one of the data mart replicates some materialized view of the other data mart. Note that, although they overlap, the bowtie and the view triangular configurations are incomparable (none is included into the other).

2.3 Transaction Model

The *transaction model* defines the properties of the transactions that access the replica copies at each node. Moreover, we assume that once a transaction is submitted for execution to a local transaction manager at a node, all conflicts are handled by the local concurrency control protocol. In our framework, we fix the properties of the transactions. We focus on three types of transactions that read or write replica copies: *update transactions*, *refresh transactions* and *queries*. All these transactions access only local data.

An update transaction is a local user transaction (i.e., executing on a single node) that updates a set of primary copies. Updates performed by an update transaction T are made visible to other transactions only after T's commitment. We denote $T_{R_1,...,R_k}$ an update transaction T that updates primary copies $R_1, ..., R_k$. We assume that no user transaction can update a materialized view.

A refresh transaction associated with an update transaction T and a node N, is composed by the serial sequence of write operations performed by T on the replica copies hold by N. We denote $RT_{r_1,...,r_k}$ a refresh transaction that updates secondary copies $r_1, ..., r_k$. For instance, if T updates R_1, S_1, and R_2, and N holds s_1 and r_2, then RT_{s_1,r_2} is the refresh transaction associated with T and N that refreshes s_1 and r_2.

Finally, *a query transaction*, noted Q, consists of a sequence of read operations on replica copies.

2.4 Propagation

The propagation parameter defines "when" the updates to a primary copy must be multicast towards the nodes storing its secondary copies. The multicast is assumed to be reliable and to preserve the global FIFO order [19] : the updates are received by the involved nodes in the order they have been multicast by the node having the primary copy.

Following [24], we focus on two types of propagation: *deferred* and *immediate*. When using a *deferred* propagation strategy, the sequence of operations of each refresh transaction associated with an update transaction T is multicast to the appropriate nodes within a single message M, after the commitment of T. When using an *immediate* propagation, each operation of a refresh transaction associated with an update transaction T is immediately multicast inside a message m, without waiting for the commitment of T.

2.5 Refreshment

The *refreshment algorithm* of a lazy master replicated system defines for each node: (i)the *triggering parameter* i.e., when a refresh transaction is started, and (ii) the *ordering parameter* i.e., the commit order of refresh transactions.

We consider three triggering modes: *deferred*, *immediate* and *wait*. The combination of a propagation parameter and a triggering mode determines a specific update propagation strategy. With a *deferred-immediate* strategy, a refresh transaction RT is submitted

for execution as soon as the corresponding message M is received by the node. With an *immediate-immediate* strategy, a refresh transaction RT is started as soon as the first message m corresponding to the first operation of RT is received. Finally, with an *immediate-wait* strategy, a refresh transaction RT is submitted for execution only after the last message m corresponding to the commitment of the update transaction associated with RT is received.

3 Consistency and Correctness Criteria

In this section, we first formally define the notion of a correct refreshment algorithm, which characterizes a refreshment algorithm that does not allow inconsistent states in a lazy master replicated system. Then for each type of configuration introduced in Section 2, we provide criteria that must be satisfied by a refreshment algorithm in order to be correct.

We now introduce useful preliminary definitions similar to those used in [16] in order to define the notion of a consistent replicated database state. We do not consider node failures, which are out of the scope of this paper. As a first requirement, we impose that any committed update on a primary copy must be eventually reflected by all its secondary copies.

Definition 3.1 *(Validity). A refreshment algorithm used in a lazy master replicated system is said valid iff any node that has a copy of a primary copy updated by a committed transaction T is guaranteed to commit the refresh transaction RT associated with T.*

Definition 3.2 *(Observable State). Let N be any node of a lazy master replicated system, the observable state of node N at local time t is the instance of the local data that reflects all and only those update and refresh transactions committed before t at node N.*

In the next definitions, we assume a global clock so that we can refer to global times in defining the notion of consistent global database state. The global clock is used for concept definition only. We shall also use the notation $I_t[N](Q)$ to denote the result of a query transaction Q run at node N at time t.

Definition 3.3 *(Quiescent State). A lazy master replicated database system is in a quiescent state at a global time t if all local update transactions submitted before t have either aborted or committed, and all the refresh transactions associated with the committed update transactions have committed.*

Definition 3.4 *(Consistent Observable State). Let N be any node of a lazy master replicated system D. Let t be any global time at which a quiescent state of D is reached. An observable state of node N at time $t_N \leq t$ is said to be consistent iff for any node N' holding a non-empty set X of replica copies hold by N and for any query transaction Q over X, there exists some time $t_{N'} \leq t$ such that $I_{t_N}[N](Q) = I_{t_{N'}}[N'](Q)$.*

Definition 3.5 *(Correct Refreshment Algorithm for a node N). A refreshment algorithm used in a lazy master replicated system D, is said to be correct for a node N of D iff it is valid and for any quiescent state reached at time t, any observable state of N at time $t_N \leq t$ is consistent.*

Definition 3.6 *(Correct Refreshment Algorithm). A refreshment algorithm used in a lazy master replicated system D, is said to be correct iff it is correct for any node N of D.*

In the following, we define correctness criteria for acyclic configurations that are sufficient conditions on the refreshment algorithm to guarantee that it is correct. The proofs of all propositions can be found in the appendix.

3.1 Global FIFO Ordering

For 1master-per-slave configurations, inconsistencies may arise if slaves can commit their refresh transactions in an order different than their corresponding update transactions. Although in 1slave-per-master configurations, every primary copy has a single associated secondary copy, the same case of inconsistency could occur between the primary and secondary copies. The following correctness criterion prevents this situation.

Definition 3.7 *(Global FIFO order). Let T_1 and T_2 be two update transactions committed by the same master or master/slave node M. If M commits T_1 before T_2, then for every node having a copy of a primary copy updated by T_1, a refresh transaction associated with T_2 can only commit after the refresh transaction associated with T_1.*

Proposition 3.1 *If a lazy master replicated system D has an acyclic configuration which is neither a bowtie nor a triangular configuration, and D uses a valid refreshment algorithm meeting the global FIFO order criterion, then this refreshment algorithm is correct.*

A similar result was shown in [7] using serializability theory.

3.2 Total Ordering

Global FIFO ordering is not sufficient to guarantee the correctness of refreshment for bowtie configurations. Consider the example in Figure 1(c), which has two master nodes, node 1 and node 2, that store relations $R(A)$ and $S(B)$, respectively. The updates performed on R by some transaction T_R: insert $R(A:a)$, are multicast towards nodes 3 and 4. In the same way, the updates performed on S by some transaction T_S: insert $S(B:b)$, are multicast towards nodes 3 and 4. With the correctness criterion of proposition 3.1, there is no ordering among the commits of refresh transactions RT_r and RT_s associated with T_R and T_S. Therefore, it might happen that RT_r commits before RT_s at node 3 and in a reverse order at node 4. In which case, a simple query transaction Q that computes $(R - S)$ could return an empty result at node 4, which is impossible at node 3. The following criterion requires that RT_r and RT_s commit in the same order at both nodes 3 and 4.

Definition 3.8 *(Total order). Let T_1 and T_2 be two committed update transactions. If two nodes commit both the associated refresh transactions RT_1 and RT_2, they both commit RT_1 and RT_2 in the same order.*

Proposition 3.2 *If a lazy master replicated system D that has a bowtie configuration but not a triangular configuration, uses a valid refreshment algorithm meeting the global FIFO order and the total order criteria, then this refreshment algorithm is correct.*

3.3 Master/slave Induced Ordering

We first extend the model presented in Section 2 to deal with materialized views as follows. From now on, we shall consider that in a master/slave node MS having a materialized view, say $V(s_1)$, any refresh transaction of s_1 is understood to encapsulate the update of some virtual copy $\hat{V}$. The actual replica copies V and v are then handled as if they were secondary copies of $\hat{V}$. Hence, we consider that the update of the virtual copy $\hat{V}$ is associated with:

- at node MS, a refresh transaction of V, noted RT_V,
- at any node S having a secondary copy of v, a refresh transaction of v, noted RT_v.

With this new modeling in mind, consider the example of Figure 2(b). Let $V(A)$ be the materialized view defined from the secondary copy $s1$. Suppose that at the initial time t_o of the system, the instance of $V(A)$ is: $\{V(A : 8)\}$ and the instance of $S(B)$ is: $\{S(B : 9)\}$. Suppose that we have two update transactions T_S and $T_{\hat{V}}$, running at nodes 1 and 2 respectively: T_S: [delete $S(B : 9)$; insert $S(B : 6)$], and $T_{\hat{V}}$: [if exists $S(B : x)$ and $x \leq 7$ then delete $V(A : 8)$; insert $V(A : 5)$]. Finally, suppose that we have the following query transaction Q over V and S, Q: [if exists $V(A : x)$ and $S(B : y)$ and $y < x$ then *bool* = *true* else *bool* = *false*], where *bool* is a variable local to Q.

Now, a possible execution is the following. First, T_S commits at node 1 and its update is multicast towards nodes 2 and 3. Then, RT_{s_1} commits at node 2. At this point of time, say t_1, the instance of s_1 is $\{s_1(B : 6)\}$. Then the update transaction $T_{\hat{V}}$ commits, afterwards the refresh transaction RT_V commits. The instance of V is $\{V(A : 5)\}$. Then at node 3, RT_v commits (the instances of v and s_2 are $\{v(A : 5)\}$ and $\{s_2(B : 9)\}$), and finally, RT_{s_2} commits (the instances of v and s_2 are $\{v(A : 5)\}$ and $\{s_2(B : 6)\}$). A quiescent state is reached at this point of time, say t_2.

However, there exists an inconsistent observable state. Suppose that Q executes at time t_1 on node 2. Then, Q will return a value *true* for *bool*. However, for any time between t_0 and t_2, the execution of Q on node 3 will return a value *false* for *bool*, which contradicts our definition of consistency.

The following criterion imposes that the commit order of refresh transactions must reflect the commit order at the master/slave node.

Definition 3.9 *(Master/slave induced order). If MS is a node holding a secondary copy s_1 and a materialized view V, then any node N_i, $i > 1$, having secondary copies s_i and v_i must commit its refresh transactions RT_{s_i} and RT_{v_i} in the same order as RT_V and RT_{s_1} commit at MS.*

Proposition 3.3 *If a lazy master replicated system D that has a view triangular configuration but not a bowtie configuration, uses a valid refreshment algorithm meeting the global FIFO order and the master/slave induced order criteria then this refreshment algorithm is correct.*

As we explained before, a configuration can be both a bowtie and a view triangular configuration. In this case, the criteria for both configurations must be enforced.

Proposition 3.4 *If a lazy master replicated system D having both a view triangular configuration and a bowtie configuration, uses a valid refreshment algorithm meeting the global FIFO order, the master/slave induced order and the total order criteria, then this refreshment algorithm is correct.*

4 Refreshment Algorithm

We start this section by presenting the system architecture of the nodes assumed by our algorithms. Then, we present our refreshment algorithm that uses a deferred update propagation strategy and prove its correctness. Finally we discuss the rationale for our algorithm.

4.1 System Architecture of Nodes

To maintain the autonomy of each node, we assume that four components are added to a regular database system, that includes a transaction manager and a query processor, in order to support a lazy master replication scheme. Figure 3 illustrates these components for a node having both primary and secondary copies. The first component, called *Replication Module*, is itself composed of three sub-components: a Log Monitor, a Propagator and a Receiver. The second component, called *Refresher*, implements a refreshment strategy. The third component, called *Deliverer*, manages the submission of refresh transactions to the local transaction manager. Finally, the last component, called *Network Interface*, is used to propagate and receive update messages (for simplicity, it is not portrayed on Figure 3). We now detail the functionality of these components:

We assume that the Network Interface provides a global FIFO reliable multicast [19] with a known upper bound [11]: messages multicast by a same node are received in the order they have been multicast. We also assume that each node has a local clock. For fairness reasons, clocks are assumed to have a bounded drift and to be ε synchronized. This means that the difference between any two correct clocks is not higher than the precision ε.

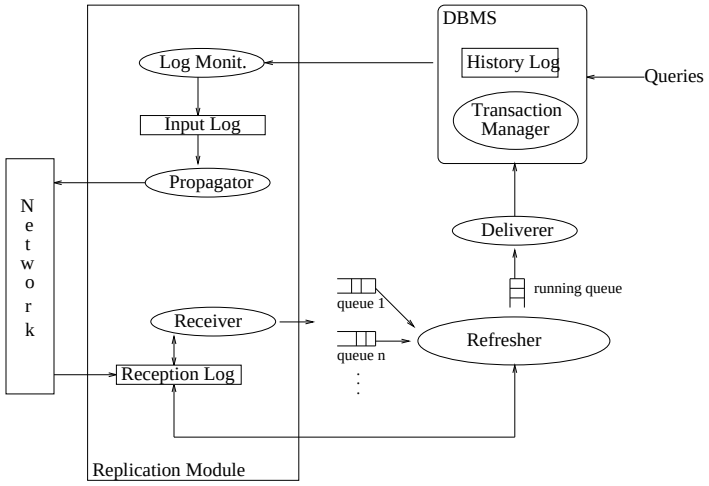

Figure 3: Architecture of a node.

The *Log Monitor* uses *log sniffing* [25, 20, 21] to extract the changes to a primary copy by continuously reading the content of a local History Log (noted H). We safely assume (see Chap. 9 of [14]) that a log record contains all the information we need such as the timestamp of a committed update transaction, and other relevant attributes that will be presented in the next section. Each committed update transaction T has a timestamp (henceforth denoted C), which corresponds to the real time value at T's commitment time. When the log monitor finds a write operation on a primary copy, it reads the corresponding log record from H and writes it into a stable storage, called *Input Log*, that is used by the Propagator. We do not deal with conflicts between the write operations on the History Log and the read operations performed by the Log Monitor since this aspect is well handled by commercial systems [21].

The *Receiver* implements update message reception. Messages coming from different masters or master/slaves are received and stored into a *Reception Log*. The receiver then reads messages from this log and stores them in FIFO *pending queues*. We denote Max, the upper bound of the time needed to multicast a message from a node and insert it into a pending queue at a receiving node. A node N has as many pending queues $q_1, ...q_n$ as masters or master/slaves nodes from which N has a secondary copy. The contents of these queues form the input to the Refresher.

The *Propagator* implements the propagation of update messages constructed from the Log Monitor. Such messages are first written into the *Input Log*. The propagator then continuously reads the *Input Log* and propagates messages through the network interface.

The *Refresher* implements the refreshment algorithm. First, it reads the contents of the pending queues, and based on its refreshment parameters, submits refresh transactions

```
Deferred-Immediate Refresher
input: pending queues q1...qn
output: running queue
variables:
    curr_M, new_M: messages from pending queues;
    timer: local reverse timer whose state is either active or inactive;
begin
        timer.state = inactive;
        curr_M = new_M = ∅;
        repeat
            on arrival of a new message or a change of timer's state to inactive do
                Step 1:
                new_M ← message with min C among top messages of q1...qn;
                Step 2:
                if new_M ≠ curr_M
                then
                        curr_M ← new_M;
                        calculate deliver_time(curr_M);
                        timer.value ← deliver_time(curr_M) − local_time
                        timer.state ← active;
                endif

            on timer.value = 0 do
                Step 3:
                write curr_M into running queue;
                dequeue curr_M from its pending queue;
                timer.state ← inactive;
        for ever
end
```

Figure 4: Deferred-immediate refreshment algorithm

by inserting them into a *running queue*. The running queue contains all ordered refresh transactions not yet entirely executed.

Finally, the *Deliverer* submits refresh transactions to the local transaction manager. It reads the content of the running queue in a FIFO order and submits each write operation as part of a refresh transaction to the local transaction manager.

4.2 Refreshment Algorithm

As described in Section 2, the refreshment algorithm has a triggering and an ordering parameters. In this section, we present the refreshment algorithm in the case of a *deferred-immediate* update propagation strategy (i.e., using an immediate triggering), and focus on the ordering parameter.

The principle of the refreshment algorithm is the following. A refresh transaction RT is committed at a slave or master/slave node (1) once all its write operations have been done, (2) according to the order given by the timestamp C of its associated update transaction, and (3) at the earliest, at real time $C + Max + \varepsilon$, which is called the deliver time, noted $deliver\_time$. Therefore, as clocks are assumed to be ε synchronized, the effects of updates on secondary copies follow the same chronological order in which their corresponding primary copies were updated.

We now detail the algorithm given in Figure 4. Each element of a pending queue is a message that contains: a sequence of write operations corresponding to a refresh transaction RT, and a timestamp C of the update transaction associated with RT. Since messages successively multicast by a same node are received in that order by the destination nodes, in any pending queue, messages are stored according to their multicast order (or commitment order of their associated update transactions).

Initially, all pending queues are empty, and $curr\_M$ and new_M are empty too. Upon arrival of a new message M into some pending queue signaled by an event, the Refresher assigns variable new_M with the message that has the smallest C among all messages in the top of all pending queues. If two messages have equal timestamps, one is selected according to the master or master/slave identification priorities. This corresponds to Step 1 of the algorithm. Then, the Refresher compares new_M with the currently hold message $curr\_M$. If the timestamp of new_M is smaller than the timestamp of $curr\_M$, then $curr\_M$ gets the value of new_M. Its deliver time is then calculated, and a local reverse timer is set with value $deliver\_time - local\_time$. This concludes Step 2 of the algorithm. Finally, whenever the timer expires its time, signaled by an event, the Refresher writes $curr\_M$ into the running queue and dequeues it from its pending queue. Each message of the running queue will yield a different refresh transaction. Note that if an update message takes Max time to reach a pending queue, it can be processed immediately by the Refresher.

4.3 Correctness of the Refreshment Algorithm

We first show that the refreshment algorithm is valid for any acceptable configuration. A configuration is said *acceptable* iff (i) it is acyclic, and (ii) if it is a triangular configuration, then it is a view triangular configuration.

Lemma 4.1 *The Deferred-immediate refreshment algorithm is valid for any acceptable configuration.*

Proof: Let us consider any node N of an acceptable configuration, having at least one primary copy. Let T be any update transaction committed by N. As we do not consider processor failures, the propagator located at node N will propagate the write operations performed by T by means of an update message using the reliable multicast. Hence the update message is received by any node involved. Since (i) the message containing the timestamp of any update transaction T is the last one related to that transaction, and (ii)

the reliable multicast preserves the global FIFO order, when a node N' receives the message containing the timestamp C of T (i.e. at the latest at time $C + Max + \varepsilon$), it has previously received all the write operations related to T and involving that node. Hence the associated refresh transaction can be committed when all its write operations are done and at the earliest at time $C + Max + \varepsilon$ □.

Lemma 4.2 *(Chronological order). The Deferred-immediate refreshment algorithm ensures for any acceptable configuration that, if T_1 and T_2 are any two update transactions committed respectively at global times t_1 and t_2 then :*

- *if $t_2 - t_1 > \varepsilon$, the timestamps C_2 for T_2 and C_1 for T_1 meet $C_2 > C_1$.*
- *any node that commits both associated refresh transactions RT_1 and RT_2, commits them in the order given by C_1 and C_2.*

Proof: Let us assume that $t_2 - t_1 > \varepsilon$. Even if the clock of the node committing T_1 is ε ahead with regard to the clock of the node committing T_2, we have $C_2 > C_1$. We now assume that we have $C_2 > C_1$ and we consider a node N that commits both RT_1 and RT_2, but RT_2 first. According to the algorithm, RT_2 is not committed before local time $C_2 + Max + \varepsilon$. At that time, if N commits RT_2 before RT_1, it means that N has not received the message related to T_1. Since clocks are ε synchronized, that message would have experienced a multicast delay higher than Max. Hence, a contradiction. □

Lemma 4.3 *The Deferred-immediate refreshment algorithm satisfies the global FIFO order criterium for any acceptable configuration.*

Proof: immediate from lemma 4.2. □

Lemma 4.4 *The Deferred-immediate refreshment algorithm satisfies the total order criteria for any acceptable configuration.*

Proof: immediate from lemma 4.2. □

Lemma 4.5 *The Deferred-immediate refreshment algorithm satisfies the master/slave induced order criteria for any acceptable configuration.*

Proof: immediate from the model extension given in Section 3.3 and from lemma 4.2. □

From the previous lemmas and the propositions of Section 3, we have:

Theorem 4.1 *The Deferred-immediate refreshment algorithm is correct for any acceptable configuration.*

4.4 Discussion

A key aspect of our algorithm is to rely on the upper bound Max on the transmission time of a message by the global FIFO reliable multicast. Therefore, it is essential to have a value of Max that is not overestimated. The computation of Max resorts to scheduling theory (e.g., see [29]). It usually takes into account four kinds of parameters. First, there is the global reliable multicast algorithm itself (see for instance [19]). Second, are the characteristics of the messages to multicast (e.g. arrival laws, size). For instance, in [12], an estimation of Max is given for sporadic message arrivals. Third, are the failures to be tolerated by the multicast algorithm, and last are the services used by the multicast algorithm (e.g. medium access protocol). It is also possible to compute an upper bound Max_i for each type i of message to multicast. In that case, the refreshment algorithm at node N waits until $max_{i \in J}$ Max_i where J is the set of message types that can be received by node N.

Thus, an accurate estimation of Max depends on an accurate knowledge of the above parameters. However, accurate values of the application dependent parameters can be obtained in performance sensitive replicated database applications. For instance, in the case of data warehouse applications that have strong requirements on freshness, certain characteristics of messages can be derived from the characteristics of the operational data sources (usually, transaction processing systems). Furthermore, in a given application, the variations in the transactional workload of the data sources can often be predicted.

In summary, the approach taken by our refreshment algorithm to enforce a total order over an algorithm that implements a global FIFO reliable multicast trades the use of a worst case multicast time at the benefit of reducing the number of messages exchanged on the network. This is a well known tradeoff. In our case, this solution brings simplicity and ease of implementation.

5 Optimizations of the Refreshment Algorithm

In this section, we present two main optimizations for the refreshment algorithm presented in Section 4. First, we show that for some configurations, the deliver time of a refresh transaction needs not to include the upper bound (Max) of the network and the clock precision (ε), thereby considerably reducing the waiting time of a refresh transaction at a slave or master/slave node. Second, we show that without sacrificing correctness, the principle of our refreshment algorithm can be combined with immediate update propagation strategies, as they were presented in [24]. Performance measurements, reported in Section 6, will demonstrate the value of this optimization.

5.1 Eliminating the Deliver Time

There are cases where the waiting time associated with the deliver time of a refresh transaction can be eliminated. For instance, consider a multinational investment bank that has traders in several cities, including New York, London, and Tokyo. These traders update a

```
Immediate_immediate Refresher
input: pending queues q_1...q_n
output: running queue
variables:
    curr_m, new_m: messages from pending queues;
    timer: local reverse timer whose state is either active or inactive;
begin
        timer.state = inactive;
        curr_m = new_m = ∅;
        repeat
            on arrival of a new message or a change of timer's state to inactive do
                if m ≠ commit
                        write m into the running queue;
                        dequeue m from its pending queue;
                else
                        new_m ← commit message with min C among top messages of q_1...q_n
                        if new_m ≠ curr_m
                        then
                           curr_m ← new_m;
                           calculate deliver_time(curr_m);
                           timer.value ← deliver_time(curr_m) − local_time
                           timer.state ← active;
                        endif
                endif

            on timer.value = 0 do
                write curr_m into running queue;
                dequeue curr_m from its pending queue;
                timer.state ← inactive;
        for ever
end
```

Figure 5: Immediate-immediate refreshment algorithm

local database of positions (securities held and quantity), which is replicated using a lazy master scheme (each site is a master for securities of that site) into a central site that warehouses the common database for all traders. The common database is necessary in order for risk management software to put limits on what can be traded and to support an internal market. A trade will be the purchase of a basket of securities belonging to several sites. In this context, a delay in the arrival of a trade notification may expose the bank to excessive risk. Thus, the time needed to propagate updates from a local site to the common database must be very small (e.g., below a few seconds).

This scheme is a 1slave-per-master configuration, which only requires a global FIFO order to ensure the correctness of its refreshment algorithm (see proposition 3.1). Since, we assume a reliable FIFO multicast network, there is no need for a refresh transaction to wait at a slave node before being executed. More generally, given an arbitrary acceptable configuration, the following proposition characterizes those slave nodes that can process refresh transactions without waiting for their deliver time.

Proposition 5.1 *Let N a node of a lazy master replicated system D. If for any node N' of D, X being the set of common replicas between N and N', we have either:*

- $cardinal(X) \leq 1$, *or*
- $\forall X_1, X_2 \in X$, *the primary copies of X_1 and X_2 are hold by the same node,*

then any valid refreshment algorithm meeting the global FIFO order criteria is correct for node N.

Proof: We proceed by contradiction assuming that an inconsistent state of N can be observed. There is a time t at which a quiescent state of D is reached. There exist a node $N' \in D$, a non-empty set X of replicas hold by both N and N', a time $t_N \leq t$ and a query transaction Q over X such that, for any time $t'_N \leq t$, we have $I_{t_N}[N](Q) \neq I_{t_{N'}}[N'](Q)$. We distinguish two cases:

- Case 1: $cardinal(X) = 1$. Let $X1$ be the unique replica of X and N'' be the node holding the primary copy of $X1$. By definition, a valid refreshment protocol ensures that any node having a secondary copy of $X1$, commits the refresh transaction associated with a committed transaction updating $X1$ at node N''. The global FIFO order criteria forces nodes N and N' to commit the refresh transactions in the same order as their associated update transactions have committed at node N''. Hence, a contradiction.
- Case 2: X contains at least two distinct replicas. In the previous case, we have shown that any secondary copy commits refresh transactions according to the commit order of their associated update transactions at the primary copy. It follows that the different results obtained by P at nodes N and N' come from a misordering of two transactions commits. Let $X1$ and $X2$ the two distinct replicas of X such that node N

commits an update/refreshment of $X1$ before an update/refreshment of $X2$ and node N' commits first the update/refreshment of $X2$ and then the update/refreshment of $X1$. By assumption, the primary copies of $X1$ and $X2$ are hold by the same node N''. The global FIFO order criteria forces nodes N and N' to reproduce the same commit order as node N''. Hence, a contradiction. □

5.2 Immediate Propagation

We assume that the Propagator and the Receiver both implement an immediate propagation strategy as specified in [24], and we focus here on the Refresher. Due to space limitations, we only present the *immediate-immediate* refreshment algorithm. We have chosen the *immediate-immediate* version because it is the one that provides the best performance compared with *deferred-immediate*, as indicated in [24].

5.2.1 Immediate-immediate Refreshment

We detail the algorithm of Figure 5. Unlike deferred-immediate refreshment, each element of a pending queue is a message m that carries an operation o of some refresh transaction, and a timestamp C. Initially, all pending queues are empty. Upon arrival of a new message m in some pending queue, signaled by an event, the Refresher reads the message and if m does not correspond to a *commit*, inserts it into the running queue. Thus, any operation carried by m other than commit can be immediately submitted for execution to the local transaction manager. If m contains a *commit* operation then new_m is assigned with the commit message that has the smallest C among all messages in the top of all pending queues. Then, new_m is compared with $curr\_m$. If new_m has a smallest timestamp than $curr\_m$, then $curr\_m$ is assigned with new_m. Afterwards, the Refresher calculates the $deliver\_time$ for $curr\_m$, and timer is set as in the *deferred-immediate* case. Finally, when the timer expires, the Refresher writes $curr\_m$ into the running queue, dequeues it from its pending queue, sets the timer to inactive and re-executes Step 1.

5.2.2 Correctness of immediate-immediate Refreshment

Like the deferred-immediate refreshment algorithm, the immediate immediate algorithm enforces refresh transactions to commit in the order of their associated update transactions. Thus, the proofs of correctness for any acceptable configuration are the same for both refreshment algorithms.

6 Performance Evaluation

In this section, we summarize the main performance gains obtained by an *immediate-immediate* refreshment algorithm against a *deferred-immediate* one. More extensive performance results are reported in [24]. We use a simulation environment that reflects as

much as possible a real replication context. We focus on a bowtie configuration which requires the use of a $Max + \varepsilon$ deliver time, as explained in Section 5.2. However, once we have fixed the time spent to reliably multicast a message, we can safely run our experiments with a single slave and several masters.

Our simulation environment is composed of *Master*, *Network*, *Slave* modules and a database server. The Master module implements all relevant capabilities of a master node such as log monitoring and message propagation. The Network module implements the most significant factors that may impact our update propagation strategies such as the delay to reliably multicast a message. The Slave module implements the most relevant components of the slave node architecture such as Receiver, Refresher and Deliverer. In addition, for performance evaluation purposes, we add the Query component in the slave module, which implements the execution of queries that read replicated data. Since, we do not consider node failures, the reception log is not taken into account. Finally, a database server is used to implement refresh transactions and query execution.

Our environment is implemented on a Sun Solaris workstation using Java/JDBC as the underlying programming language. We use sockets for inter-process communication and Oracle 7.3 to implement refresh transaction execution and query processing. For simulation purposes, each write operation corresponds to an UPDATE command that is submitted to the server for execution.

6.1 Performance Model

The metrics used to compare the two refreshment algorithms is given by the freshness of secondary copies at the slave node. More formally, given a replica X, which is either a secondary or a primary copy, we define $n(X,t)$ as the number of committed update transactions on X at global time t. We assume that update transactions can have different sizes but their occurence is uniformly distributed over time. Using this assumption, we define the degree freshness of a secondary copy r at global time t as:

$$f(r,t) = n(r,t)/n(R,t);$$

Therefore, a degree of freshness close to 0 means bad data freshness while close to 1 means excellent. The mean degree of freshness of r at a global time T is defined as:

$$mean_f = 1/T \int_0^T f(r,t)dt$$

We now present the main parameters for our experimentations summarized in Table 1. We assume that the mean time interval between update transactions, noted λ_t, as reflected by the history log of each master, is bursty. Updates are done on the same attribute (noted *attr*) of a different tuple. We focus on *dense* update transactions, i.e., transactions with a small time interval between each two writes. We define two types of update transactions.

Small update transactions have size 5 (i.e., 5 write operations), while long transactions have size 50. We define four scenarios in which the proportion of long transactions, noted ltr, is set respectively to 0, 30, 60, and 100. Thus, in a scenario where ltr = 30, 30 % of the executed update transactions are long. Finally, we define an abort transaction ratio, noted abr, of 0, 5%, 10%, 20%, that corresponds to the percentage of transactions that abort in an experiment. Furthermore, we assume that a transaction abort always occurs after half of its execution. For instance $abr = 10\%$ means that 10% of the update transactions abort after the execution of half of their write operations.

Network delay is calculated by $\delta+t$, where δ is the time between the insertion of a message in the input queue of the Network module and the multicast of the message by that module, and t is the reliable multicast time of a message until its insertion in the pending queue of the Refresher. Concerning the value of t used in our experiments, we have a short message multicast time, noted t_{short}, which represents the time needed to reliably multicast a single log record. In addition, we consider that the time spent to reliably multicast a sequence of log records is linearly proportional to the number of log records it carries. The network overhead delay, δ, takes into account the time spent in the input queue of the Network, it is implicity modeled by the system overhead to read from and write to sockets. The *Total propagation time* (noted t_p) is the time spent to reliably multicast all log records associated with a given transaction. Thus, if n represents the size of the transaction with immediate propagation, we have $t_p = n \times (\delta + t_{short})$, while with deferred propagation, we have $t_p = (\delta + n \times t_{short})$. Network contention occurs when δ increases due to the increase of network traffic. In this situation, the delay introduced by δ may impact the total propagation time, especially with immediate propagation. Finally, when $ltr > 0$, the value of Max is calculated using the maximun time spent to reliably multicast a long transaction $(50 * t_{short})$. On the other hand, when $ltr = 0$, the value of Max is calculated using the maximun time spent to reliably multicast a short transaction $(5 * t_{short})$.

The refresh transaction execution time is influenced by the existence of possible conflicting queries that read secondary copies at the slave node. Therefore, we need to model queries. We assume that the mean time interval between queries is low, and the number of data items read is small (fixed to 5). We fix a 50% conflict rate for each secondary copy, which means that each refresh transaction updates 50% of the tuples of each secondary copy that are read by a query.

To measure the mean degree of freshness, we use the following variables. Each time an update transaction commits at a master, variable $version\_master$ for that master, is incremented. Similarly, each time a refresh transaction commits at the slave, variable $version\_slave$, is incremented. Whenever a query conflicts with a refresh transaction we measure the degree of freshness.

6.2 Experiments

We present three experiments. The results are average values obtained from the execution of 40 update transactions. The first experiment shows the mean degree of freshness obtained

Parameters	*Definition*	*Values*
λ_t	mean time interval between Trans.	*bursty: (mean* $= 200ms$*)*
λ_q	mean time interval between Queries	*low (mean* $= 15s$*)*
$nbmaster$	Number of Master nodes	1 to 8
$\|Q\|$	Query Size	5
$\|RT\|$	Refresh Transaction Size	5; 50
ltr	Long Transaction Ratio	0; 30%; 60%; 100%
abr	Abort Ratio	0; 5%; 10%; 20%
t_{short}	Multicast Time of a single record	20ms and 100ms

Table 1: Performance Parameters

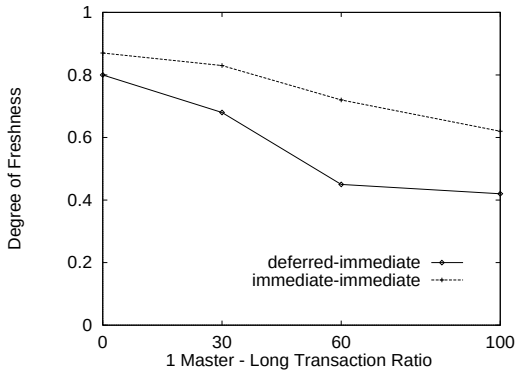

Figure 6: Bursty Workload - Mean Degree of Freshness

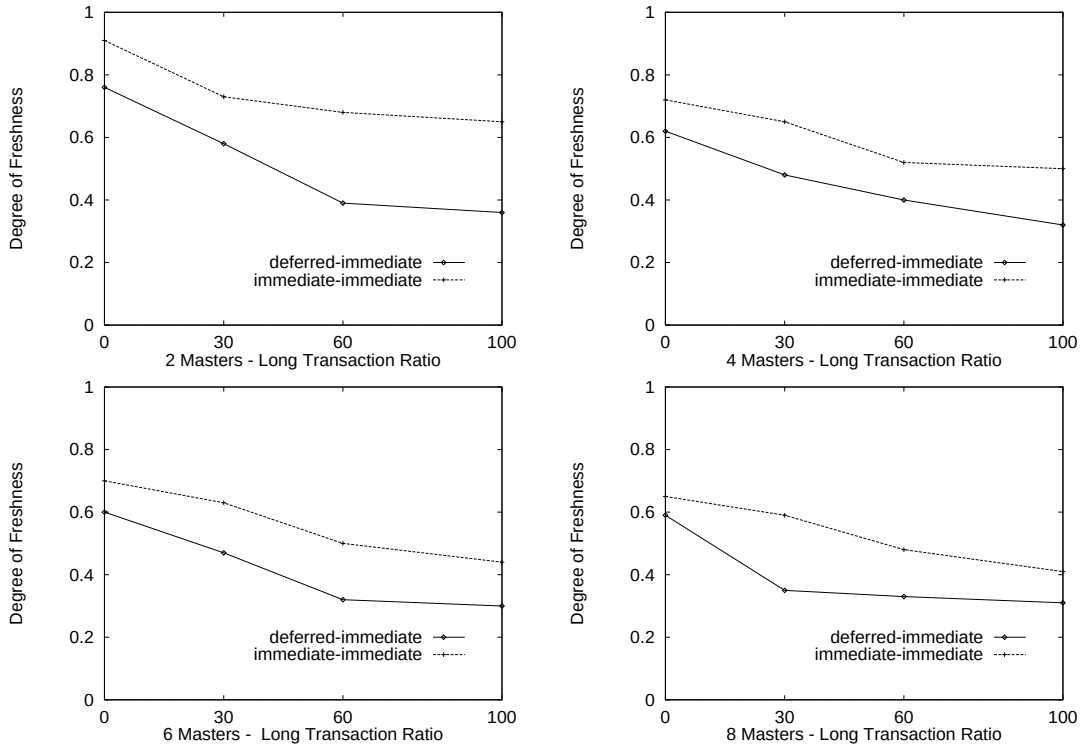

Figure 7: Bursty Workload - Mean Degree of Freshness

for the *bursty* workload. The second experiment studies the impact on freshness when update transactions abort. In the third experiment, we verify the freshness improvement of each strategy when the network delay to propagate a message increases.

We now summarize and discuss the major observations of our experiments. As depicted in Figure 6, when $ltr = 0$ (short transactions), the mean degree of freshness is already impacted because on average, $\lambda_t < t_p$. Therefore, during T_i's update propagation, $T_{i+1}, T_{i+2}...T_{i+n}$ may be commited. Notice that even with the increase of network contention, *immediate-immediate* yields a better mean degree of freshness. With 2, 4, and 8 masters, the results of *immediate-immediate* are much better than those of *deferred-immediate*, as *ltr* increases (see Figure 7). For instance, with 4 masters with $ltr = 30$, the mean degree of freshness is 0.62 for *immediate-immediate* and 0.32 for *deferred-immediate*. With 6 masters and $ltr = 60$, the mean degree of freshness is 0.55 for *immediate-immediate*, and 0.31 for *deferred-immediate*. In fact, *immediate-immediate* always yields the best mean degree of freshness even with network contention due to the parallelism of log monitoring, propagation, and refreshment.

With *immediate-immediate*, the mean query response time may be seriously impacted because each time a query conflicts with a refresh transaction, it may be blocked during

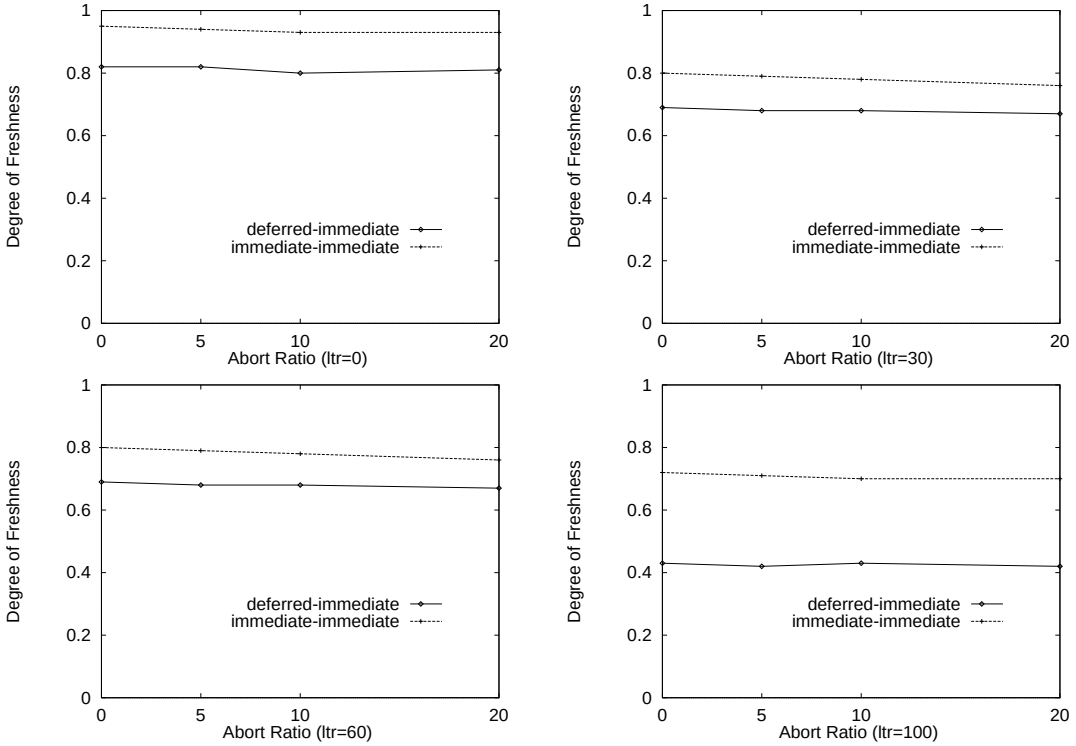

Figure 8: Bursty Workload - *Abort* Effects

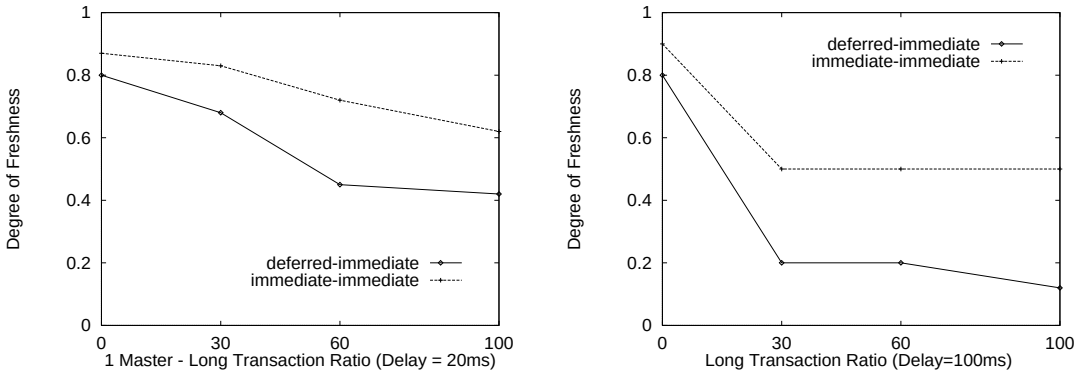

Figure 9: Increase of Network Delay - Mean Degree of Freshness

a long period of time since the propagation time may be added to the refresh transaction execution time. When $\lambda_q >> \lambda_t$, the probability of conflicts is quite high. There, the network delay to propagate each operation has an important role and the higher its value, the higher are the query response times in conflict situations. However, we verified that the use of a multiversion protocol on the slave node may significantly reduces query response times, without a significant decrease in the mean degree of freshness.

The *abort* of an update transaction with *immediate-immediate* does not impact the mean degree of freshness since the delay introduced to undo a refresh transaction is insignificant compared to the propagation time. As shown in Figure 8, for $ltr = 0$ and various values of abr (5,10,20), the decrease of freshness introduced by update transactions that abort with *immediate-immediate* is insignificant. In the worst case, it achieves 0.2. This behavior is the same for other values of ltr (30, 60, 100).

Finally, the improvements brought by *immediate-immediate* are more significant when the network delay to propagate a single operation augments. Figure 9 compares the freshness results obtained when $\delta = 100ms$ and $\delta = 20ms$. For instance, when $\delta = 20$ and $ltr = 100$ *immediate-immediate* improves 1.1 times better than *deferred-immediate* and when $\delta = 100$, *immediate-immediate* improves 5 times better. This clearly shows the advantages of having tasks being executed in parallel.

7 Related Work

The closest work to ours is in [7]. Essentially, using our terminology, the authors show that for any strongly acyclic configuration a refreshment algorithm (called there spu), which enforces a global FIFO ordering, guarantees a global serializability property, which is similar to our noton of correction. Their result is analogous to our Proposition 3.1. They also propose an algorithm, which assigns, when it is possible, a site to each primary copy so that the resulting configuration is strongly acyclic. However, no algorithm is provided to refresh secondary copies in the cases of non strongly acyclic configurations.

Much work has been devoted to the maintenance of integrity constraints in federated or distributed databases, including the case of replicated databases [5, 18, 9, 16]. These papers propose algorithms and protocols to prevent the violation of certain kind of integrity constraints by local transactions. However, their techniques are not concerned with the consistent refreshment of replicas. On the contrary, by maintaining the consistency of replicas, we provide the basis for the deployment of distributed integrity checking algorithms in replicated databases.

Our relationship to papers that proposed weaker consistency criteria was already presented in Section 1.

In [25], the authors describe a lazy group replication scheme in which the update propagation protocol applies updates to replicated data in their arrival order, possibly restoring inconsistencies when arrivals violate the timestamp ordering of transactions. The major dif-

ference with our work is that they achieve consistency by undoing and re-executing updates which are out-of-order, whereas we do not allow inconsistent database states.

The timestamp message delivery protocol in [13] implements eventual delivery for a lazy group replication scheme [15]. It uses periodic exchange of messages between pairs of servers that propagate messages to distinct groups of master nodes. At each master node incoming messages are stored in a history log (as initially proposed in [20]) and later delivered to the application in a defined order. Eventual delivery is not appropriate in our framework since we are interested in improving data freshness.

The goal of epidemic algorithms [1, 28] is to ensure that all replicas of a single data item converge to a single final value in a lazy group replication scheme. Updates are executed locally at any node. Later, nodes communicate to exchange up-to-date information. In our approach, updates are propagated from each primary copy towards all its secondary copies instead.

In [27], the authors explore the use of different variants of broadcast protocols to perform update propagation in replicated databases. Their correctness criteria is serializability. In contrast, in our work, we define several correctness criterias and fix a multicast protocol.

Formal concepts for specifying coherency conditions in a replicated distributed database have been introduced in [10]. The authors focus on a *deferred-immediate* update propagation strategy and propose concepts for computing an independent measure of relaxation, called *coherency index*. Their concept of *version* which considers the number of updates occuring on a primary copy still not computed in a secondary copy is closely related to our notion of freshness. However, the analytical model proposed in [22] could be used in conjonction with real performance measurements to further explore the performance tradeoff of our algorithm. Complementary to our work, their measurements could be adapted to evaluate our immediate strategy.

8 Conclusion

In a lazy master replicated system, a transaction can commit after updating one replica copy (primary copy) at some node. The updates are propagated towards the other replicas (secondary copies), and these replicas are refreshed in separate refresh transactions.

In this paper, we proposed refreshment algorithms which address the central problem of maintaining replicas' consistency. That is, an observer of a set of replicas at some node never observes a state which is never seen by another oberver of the same set of replicas at another node.

This paper has three major contributions. Our first contribution is a formal definition of (i) the notion of correct refreshment algorithm and (ii) correctness criteria for any acceptable configuration (e.g. 1master-per-slave, 1 slave-per-master, bowtie and view triangular, ...).

Our second contribution is an algorithm meeting these correctness criteria for any acceptable configuration. This algorithm can be easily implemented over an existing database system. It is based on a deferred update propagation, and it delays the execution of a refresh transaction until its deliver time.

Our third contribution concerns optimizations of the refreshment algorithm in order to improve the algorithm efficiency. This efficiency is evaluated by the data freshness. The first optimization avoids waiting until the deliver time in certain cases. For any configuration, we characterized the nodes that do not need to wait. The second optimization improves data freshness by using *immediate-immediate* update propagation strategy. This strategy allows parallelism between the propagation of updates and the execution of the associated refresh transactions.

Finally, a performance evaluation shows the main performance gains obtained by our *immediate-immediate* refreshment algorithm against a *deferred-immediate* using Oracle 7.3. Our results show that the *immediate-immediate* strategy always yields the best mean degree of freshness for a bursty workload. In addition, the *abort* of an update transaction with *immediate-immediate* strategy does not impact the mean degree of freshness since the delay introduced to undo a refresh transaction is insignificant compared to the propagation time. Finally, the improvements brought by *immediate-immediate* strategy are more significant (e.g. by a factor 5) when the network delay to propagate a single operation augments.

In a future work, we plan to study how to determine an accurate bound Max on the response time of a global FIFO reliable multicast in some typical data warehouse applications.

9 Appendix

We first consider each replica copy individually and show that at each replica copy, updates are committed in the order they have been committed at the primary copy.

Lemma 9.1 *In a lazy master replicated system D, using a valid refreshment protocol meeting the global FIFO order criterion, let t be any global time at which a quiescent state of D is reached. For any node N, for any replica copy X hold by N, for any node N' holding X, for any query transaction Q over the only replica copy X, for any time $t_N \leq t$, there exists a time $t_{N'} \leq t$ such that $I_{t_N}[N](Q) = I_{t_{N'}}[N'](Q)$.*

Proof: We proceed by contradiction. We assume that there is a time $t_N \leq t$ such that for any time $t_{N'} \leq t$ we have : $I_{t_N}[N](Q) \neq I_{t_{N'}}[N'](Q)$. We distinguish two cases :

- either N holds the primary copy of X. Hence the query over X at node N reflects all the update transactions committed before t_N. According to the validity property, all the associated refresh transactions will be committed at nodes having a secondary copy of X. Moreover the global FIFO order criterion forces the refresh transactions to be committed in the order of their associated update transactions. Hence a contradiction.
- or N holds a secondary copy of X. If N' holds the primary copy of X, by analogy with the previous case, we obtain a contradiction. If now N' holds a secondary copy of X, then by the validity property and by the global FIFO order criterion, all the nodes having a secondary copy of X must reflect all the updates transactions committed before t_N and in the order they have been committed on the primary copy of X. Hence a contradiction. □

We now show that if a query transaction over a common set of replicas gives different results at two nodes, then there exist two replicas $X1$ and $X2$ in X such that their updates have been committed in a different order by nodes N and N'.

Lemma 9.2 *In a lazy master replicated system D, using a valid refreshment protocol meeting the global FIFO order criterion, let t be any global time at which a quiescent state of D is reached. If there are nodes N and N', a non-empty set X of replica copies hold by N and N', a query transaction Q over X, a time $t_N \leq t$ such that for any time $t_{N'} \leq t$ $I_{t_N}[N](Q) \neq I_{t_{N'}}[N'](Q)$, then there are two distinct replicas $X1$ and $X2$ in X such that their updates/refreshes have been committed in a different order by nodes N and N'.*

Proof: From lemma 9.1, this is impossible if the cardinal of X is one. Hence we assume that X contains at least two distinct replicas. If the results of Q over X differ at nodes N and N', it means that the transactions having updated/refreshed the replicas in X have committed in a different order at nodes N and N'. Let $X1$ and $X2$ with $X1 \neq X2$ be the replicas in X such that node N commits an update/refresh of $X1$ before an update/refresh of $X2$, and node N' commits an update/refresh of $X2$ before an update/refresh of $X1$. □

We now consider all the possible cases for two nodes of an acyclic configuration, holding both at least two replica copies.

Lemma 9.3 *In an acyclic configuration of a lazy master replicated system D, for any two distinct nodes N and N' holding both two distinct replica copies $X1$ and $X2$, the only possible cases are:*

1. *either N has the primary copies of both $X1$ and $X2$; N' is a slave of N;*
2. *or N' has the primary copies of both $X1$ and $X2$; N is a slave of N';*
3. *or both N and N' have secondary copies of both $X1$ and $X2$.*
4. *or N has the primary copy of $X1$ and a secondary copy of $X2$; N' has secondary copies of both $X1$ and $X2$.*
5. *or N' has the primary copy of $X1$ and a secondary copy of $X2$; N has secondary copies of both $X1$ and $X2$.*
6. *or N has the primary copy of $X2$ and a secondary copy of $X1$; N' has secondary copies of both $X1$ and $X2$.*
7. *or N' has the primary copy of $X2$ and a secondary copy of $X1$; N has secondary copies of both $X1$ and $X2$.*

Proof: We consider all the possible cases for two distinct nodes N and N' holding both a replica copy of $X1$ and a replica copy of $X2$ with $X1 \neq X2$. We have 16 possible cases for the attribution of the primary/secondary copy of $X1$ and $X2$ to N and N'. Each case can be coded with four bits with the following meaning:

- the first bit is one if N holds the primary copy of $X1$ and zero otherwise;
- the second bit is one if N holds the primary copy of $X2$ and zero otherwise;
- the third bit is one if N' holds the primary copy of $X1$ and zero otherwise;
- the fourth bit is one if N' holds the primary copy of $X2$ and zero otherwise;

Among them, seven are impossible, because for any replica, only one node holds the primary copy. Hence, the impossible cases are 1010, 1110, 1011, 1111, 0101, 0111, 1101.

We now prove that the cases 1001 and 0110 are impossible. Let us consider the case 1001 where N holds the primary copy of $X1$ and N' holds the primary copy of $X2$. We then have N is a slave of N' (because of $X2$) and N' is a slave of N (because of $X1$). The configuration is then cyclic: a contradiction with our assumption. By analogy, the case 0110 is impossible.

Hence there are 16-7-2=7 only possible cases, which are given in the lemma. □

We can now prove the following proposition:

Proposition 9.1 *If a lazy master replicated system D has an acyclic configuration which is neither a bowtie nor a triangular configuration and D uses a valid refreshment algorithm meeting the global FIFO order criterion, then this refreshment algorithm is correct.*

Proof: We proceed by contradiction. There exist a node N and a node N' with $N \neq N'$ such that X the set of replica hold by both N and N' is non empty, there exist a query transaction Q over X and a time $t_N \leq t$ such that $\forall t'_N \leq t$, we have $I_{t_N}[N](Q) \neq I_{t_{N'}}[N'](Q)$. From lemma 9.2, there exist two distinct replicas $X1$ and $X2$ in X such that their updates have been committed in a different order by nodes N and N'.

We will now consider all the cases given by lemma 9.3.

1. either N has the primary copies of both $X1$ and $X2$; N' is a slave of N; The global FIFO order criterion enforces N' to commit the refresh transactions in the order their associated update transactions have been committed by N. Hence a contradiction.
2. or N' has the primary copies of both $X1$ and $X2$; N is a slave of N'; This is the symmetrical case of the previous one. We then obtain a contradiction.
3. or both N and N' have secondary copies of both $X1$ and $X2$. Notice that this case is impossible in a 1slave-per-master configuration (both N and N' would be slave of the node holding the primary copy of $X1$). Let $N1$ be the node holding the primary copy of $X1$. Let $N2$ be the node holding the primary copy of $X2$. Since the configuration is not a bowtie configuration, we necessarily have $N1 = N2$. The global FIFO order criterion enforces both N and N' to commit the refresh transactions in the order their associated update transactions have been committed by $N1 = N2$. Hence a contradiction.

4. or N has the primary copy of $X1$ and a secondary copy of $X2$; N' has secondary copies of both $X1$ and $X2$. There exists a node $N2$ holding the primary copy of $X2$ such that N and N' are slaves of $N2$ and N' is slave of N. This case and all the following ones would lead to a triangular configuration. Hence, a contradiction. □

Proposition 9.2 *If a lazy master replicated system D that has a bowtie configuration but not a triangular configuration, uses a valid refreshment algorithm meeting the global FIFO order and the total order criteria, then this refreshment algorithm is correct.*

Proof:
We proceed by contradiction. There exist a node N and a node N' with $N \neq N'$ such that X the set of replica hold by both N and N' is non empty, there exist a query program P over X and a time $t_N \leq t$ such that $\forall t'_N \leq t$, we have $I_{t_N}[N](P) \neq I_{t_{N'}}[N'](P)$. From lemma 9.2, there exist two distinct replicas $X1$ and $X2$ in X such that their updates have been committed in a different order by nodes N and N'.
We will now consider all the cases given by lemma 9.3.

1. either N has the primary copies of both $X1$ and $X2$; N' is a slave of N; The global FIFO order criterion enforces N' to commit the refresh transactions in the order their associated update transactions have been committed by N. Hence a contradiction.

2. or N' has the primary copies of both $X1$ and $X2$; N is a slave of N'; This is the symmetrical case of the previous one. We then obtain a contradiction.

3. or both N and N' have secondary copies of both $X1$ and $X2$. Let $N1$ be the node holding the primary copy of $X1$. Let $N2$ be the node holding the primary copy of $X2$. The total order criterion enforces both N and N' to commit the refresh transactions on $X1$ and $X2$ in the same order. Hence, a contradiction.

4. or N has the primary copy of $X1$ and a secondary copy of $X2$; N' has secondary copies of both $X1$ and $X2$. There exists a node $N2$ holding the primary copy of $X2$ such that N and N' are slaves of $N2$ and N' is slave of N. This case and all the following ones would lead to a triangular configuration. Hence, a contradiction. □

Proposition 9.3 *If a lazy master replicated system D that has a view triangular configuration but not a bowtie configuration, uses a valid refreshment algorithm meeting the global FIFO order and the master/slave induced order criteria, then this refreshment algorithm is correct.*

Proof:
We proceed by contradiction. There exist a node N and a node N' with $N \neq N'$ such that X the set of replica hold by both N and N' is non empty, there exist a query program

P over X and a time $t_N \leq t$ such that $\forall t'_N \leq t$, we have $I_{t_N}[N](P) \neq I_{t_{N'}}[N'](P)$. >From lemma 9.2, there exist two distinct replicas $X1$ and $X2$ in X such that their updates have been committed in a different order by nodes N and N'.

We will now consider all the cases given by lemma 9.3.

1. either N has the primary copies of both $X1$ and $X2$; N' is a slave of N; The global FIFO order criterion enforces N' to commit the refresh transactions in the order their associated update transactions have been committed by N. Hence a contradiction.

2. or N' has the primary copies of both $X1$ and $X2$; N is a slave of N'; This is the symmetrical case of the previous one. We then obtain a contradiction.

3. or both N and N' have secondary copies of both $X1$ and $X2$. Let $N1$ be the node holding the primary copy of $X1$. Let $N2$ be the node holding the primary copy of $X2$. Since the configuration is not a bowtie configuration, we necessarily have $N1 = N2$. The global FIFO order criterion forces both N and N' to commit the refresh transactions on $X1$ and $X2$ in the order their associated update transactions have been committed by $N1 = N2$. Hence, a contradiction.

4. or N has the primary copy of $X1$ and a secondary copy of $X2$; N' has secondary copies of both $X1$ and $X2$. There exists a node $N2$ holding the primary copy of $X2$ such that N and N' are slaves of $N2$ and N' is slave of N. As the configuration is a view triangular one, $X1$ is a materialized view from local secondary copies. The master/slave induced order criterion enforces nodes N and N' to commit the refresh transactions of $X1$ and $X2$ in the same order. Hence, a contradiction. □

Proposition 9.4 *If a lazy master replicated system D having both a view triangular configuration and a bowtie configuration, uses a valid refreshment algorithm meeting the global FIFO order, the master/slave induced order and the total order criteria then this refreshment algorithm is correct.*

Proof: We proceed by contradiction. There exist a node N and a node N' with $N \neq N'$ such that X the set of replica hold by both N and N' is non empty, there exist a query program P over X and a time $t_N \leq t$ such that $\forall t'_N \leq t$, we have $I_{t_N}[N](P) \neq I_{t_{N'}}[N'](P)$. From lemma 9.2, there exist two distinct replicas $X1$ and $X2$ in X such that their updates have been committed in a different order by nodes N and N'.

We will now consider all the cases given by lemma 9.3.

1. either N has the primary copies of both $X1$ and $X2$; N' is a slave of N; The global FIFO order criterion forces N' to commit the refresh transactions in the order their associated update transactions have been committed by N. Hence a contradiction.

2. or N' has the primary copies of both $X1$ and $X2$; N is a slave of N'; This is the symmetrical case of the previous one. We then obtain a contradiction.

3. or both N and N' have secondary copies of both $X1$ and $X2$. Let $N1$ be the node holding the primary copy of $X1$. Let $N2$ be the node holding the primary copy of $X2$. The total order criterion forces both N and N' to commit the refresh transactions on $X1$ and $X2$ in the same order. Hence, a contradiction.

4. or N has the primary copy of $X1$ and a secondary copy of $X2$; N' has secondary copies of both $X1$ and $X2$. There exists a node $N2$ holding the primary copy of $X2$ such that N and N' are slaves of $N2$ and N' is slave of N. As the configuration is a view triangular one, $X1$ is a materialized view from local secondary copies. The master/slave induced order criterion enforces nodes N and N' to commit the refresh transactions of $X1$ and $X2$ in the same order. Hence, a contradiction.

5. this case and all the following ones are symmetrical to the previous one. □

References

[1] D. Agrawal and G. Alonso and A. El Abbadi and I. Stanoi,*Exploiting Atomic Broadcast in Replicated Databases*, EURO-PAR Int. Conf. on Parallel Processing, August 1997.

[2] R. Alonso and D. Barbara and H. Garcia-Molina, *Data Caching Issues in an Information Retrieval System*, ACM Transactions on Database Systems, 15(3):359-384, September 1990.

[3] G. Alonso and A. Abbadi, *Partitioned Data Objects in Distriduted Databases*,Distributed and Parallel Databases, 3(1):5-35, 1995.

[4] P.A. Bernstein and E. Newcomer, *Transaction Processing*, Morgan Kaufmann, 1997.

[5] S. Ceri and J. Widom, *Managing Semantic Heterogeneity with production rules and persistent queues*, Int. Conf. on VLDB, 1993.

[6] S. Chaudhuri and U. Dayal, *An Overview of Data Warehousing and OLAP Technology*, ACM SIGMOD Record, 26(1), March 1997.

[7] P. Chundi and D. J. Rosenkrantz and S. S. Ravi, *Deferred Updates and Data Placement in Distributed Databases*, Int. Conf. on Data Engineering (ICDE), Louisiana, February 1996.

[8] A. Downing and I. Greenberg and J. Peha, *OSCAR: A System for Weak-Consistency Replication*, Int. Workshop on Management of Replicated Data, pages26-30, Houston, November 1990.

[9] L. Do and P. Drew, *The Mangement of Interdependent Asynchronous Transactions in Heterogeneous Database Environments*, Int. Conf, on Database Systems for Advanced Applications, 1995.

[10] R. Gallersdorfer and M. Nicola, *Improving Performance in Replicated Databases through Relaxed Coherency*, Int. Conf. on VLDB, Zurich, September 1995.

[11] L. George and P. Minet, *A FIFO worst case analysis for a hard real-time distributed problem with consistency constraints*, Int. Conf. on Distributed Computing Systems (ICDCS97), Baltimore, May 1997.

[12] L. George and P. Minet, *A uniform reliable multicast protocol with guaranteed responses times*, ACM SIGPLAN workshop on Languages Compilers and Tools for Embedded Systems, Montreal, June 1998.

[13] R. Goldring, *Weak-Consistency Group Communication and Membership*, PhD Thesis - University of Santa Cruz, 1992.

[14] J.N. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan-Kaufmann, 1993.

[15] J. Gray and P. Helland and P. O'Neil and D. Shasha, *The Danger of Replication and a Solution*, ACM SIGMOD Int. Conf on Management of Data, Montreal, June 1996.

[16] P. Grefen and J. Widom, *Protocols for Integrity Constraint Checking in Federated Databases*, Distributed and Parallel Databases, 54,October 1997.

[17] P. S. Group, *The Data Mart Option*, Open Information Systems, 11(7), 1996.

[18] H. Gupta and I.S. Mumick and V.S. Subrahmanian, *Maintaining Views Incremently*, ACM SIGMOD Int. Conference on Management of Data, Washington, DC, May 1993.

[19] V. Hadzilacos and S. Toueg, *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*, Technical Report TR 94-1425, Dept. of Computer Science, Cornell University, Ithaca, NY 14853, 1994.

[20] B. Kahler and O. Risnes, *Extending Logging for Database Snapshot Refresh*, Int. Conf on VLDB, Brighton, September 1987.

[21] A. Moissis, *Sybase Replication Server: A Pratical Architecture for Distributing and Sharing Corporate Information*, 1996.

[22] M. Nicola and M. Jarke, *Increasing the Expressiveness of Analytical Performance Models for Replicated Databases*, Int. Conference on Database Theory (ICDT), Jerusalem, January 1999.

[23] M. Tamer Özsu and P. Valduriez, *Principles of Distributed Database Systems*, 2nd Edition, Prentice Hall, 1999.

[24] E. Pacitti and E. Simon and R. de Melo, *Improving Data Freshness in Lazy Master Schemes*, Int. Conf. on Distributed Computing Systems (ICDCS98), Amsterdam, May 1998.

[25] S. K. Sarin and C. W. Kaufman and J. E. Somers, *Using History Information to Process Delayed Database Updates*, Int. Conf. on VLDB, Kyoto, August 1986.

[26] A. Sheth and M. Rusinkiewicz, *Management of Interdependent Data: Specifying Dependency and Consistency Requirements*, Int. Workshop on Management of Replicated Data, Houston, November 1990.

[27] I. Stanoi and D. Agrawal and A. El Abbadi, *Using Broadcast Primitives in Replicated Databases*, Int. Conf. on Distributed Computing Systems (ICDCS98), Amsterdam, May 1998.

[28] D. B. Terry and M. M. Theimer and K. Petersen and A. J. Demers and M. J. Spreitzer and C. H. Hauser, *Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System*, Symposium on Operating System Principles (SIGOPS), Colorado, December 1995.

[29] K. Tindell and J. Clark, *Holistic schedulability analysis for distributed hard real-time systems*, Microprocessors and Microprogramming, 40, 1994.

[30] Y. Zhuge and H. Garcia-Molina and J. L. Wiener, *View Maintenance in a Warehouse Environment*, ACM SIGMOD Int. Conf. on Management of Data, 1995.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399